<https://leetcode.com/problems/longest-common-subsequence/>

Given two strings text1 and text2, return *the length of their longest* ***common subsequence****.* If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde".

A **common subsequence** of two strings is a subsequence that is common to both strings.

**Example 1:**

Input: text1 = "abcde", text2 = "ace"

Output: 3

Explanation: The longest common subsequence is "ace" and its length is 3.

**Example 2:**

Input: text1 = "abc", text2 = "abc"

Output: 3

Explanation: The longest common subsequence is "abc" and its length is 3.

**Example 3:**

Input: text1 = "abc", text2 = "def"

Output: 0

Explanation: There is no such common subsequence, so the result is 0.

**Constraints:**

1 <= text1.length, text2.length <= 1000

text1 and text2 consist of only lowercase English characters.

**Why might we want to solve the longest common subsequence problem?**

**Refer to**

<https://leetcode.com/problems/longest-common-subsequence/solutions/436719/python-very-detailed-solution-with-explanation-and-walkthrough-step-by-step/>

File comparison. The Unix program "diff" is used to compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed. In this instance of the problem we should think of each line of a file as being a single complicated character in a string.

**Attempt 1: 2023-05-21**

**Solution 1: Native DFS (10 min, TLE)**

class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        return helper(text1, text2, 0, 0);

    }

    private int helper(String text1, String text2, int i, int j) {

        if(i == text1.length() || j == text2.length()) {

            return 0;

        }

        if(text1.charAt(i) == text2.charAt(j)) {

            return 1 + helper(text1, text2, i + 1, j + 1);

        } else {

            return Math.max(helper(text1, text2, i + 1, j), helper(text1, text2, i, j + 1));

        }

    }

}

Time Complexity : O(2^N)

Space Complexity : O(2^N)

**Java Implementation**

**Refer to**

<https://leetcode.com/problems/longest-common-subsequence/solutions/590781/from-brute-force-to-dp/>

class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        return longestCommonSubsequence(text1, text2, 0, 0);

    }

    private int longestCommonSubsequence(String text1, String text2, int i, int j) {

        if (i == text1.length() || j == text2.length())

            return 0;

        if (text1.charAt(i) == text2.charAt(j))

            return 1 + longestCommonSubsequence(text1, text2, i + 1, j + 1);

        else

            return Math.max(

                longestCommonSubsequence(text1, text2, i + 1, j),

                longestCommonSubsequence(text1, text2, i, j + 1)

            );

    }

}

**Time Complexity**

**Refer to**

<https://leetcode.com/problems/longest-common-subsequence/solutions/436719/python-very-detailed-solution-with-explanation-and-walkthrough-step-by-step/>

If the two strings have no matching characters, so the last line always gets executed, the the time bounds are binomial coefficients, which (if m=n) are close to 2^n.

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        Integer[][] memo = new Integer[text1.length()][text2.length()];

        return helper(text1, text2, 0, 0, memo);

    }

    private int helper(String text1, String text2, int i, int j, Integer[][] memo) {

        if(i == text1.length() || j == text2.length()) {

            return 0;

        }

        if(memo[i][j] != null) {

            return memo[i][j];

        }

        if(text1.charAt(i) == text2.charAt(j)) {

            return memo[i][j] = 1 + helper(text1, text2, i + 1, j + 1, memo);

        } else {

            return memo[i][j] = Math.max(helper(text1, text2, i + 1, j, memo), helper(text1, text2, i, j + 1, memo));

        }

    }

}

Time Complexity : O(NM)

Space Complexity : O(NM)

**Refer to**

<https://leetcode.com/problems/longest-common-subsequence/solutions/590781/from-brute-force-to-dp/>

We might use memoization to overcome overlapping subproblems.Since there are two changing values, i.e. i and j in the recursive function longestCommonSubsequence, we might apply a two-dimensional array.

class Solution {

    private Integer[][] dp;

    public int longestCommonSubsequence(String text1, String text2) {

        dp = new Integer[text1.length()][text2.length()];

        return longestCommonSubsequence(text1, text2, 0, 0);

    }

    private int longestCommonSubsequence(String text1, String text2, int i, int j) {

        if (i == text1.length() || j == text2.length())

            return 0;

        if (dp[i][j] != null)

            return dp[i][j];

        if (text1.charAt(i) == text2.charAt(j))

            return dp[i][j] = 1 + longestCommonSubsequence(text1, text2, i + 1, j + 1);

        else

            return dp[i][j] = Math.max(

                longestCommonSubsequence(text1, text2, i + 1, j),

                longestCommonSubsequence(text1, text2, i, j + 1)

            );

    }

}

**Time Complexity**

**Refer to**

<https://leetcode.com/problems/longest-common-subsequence/solutions/436719/python-very-detailed-solution-with-explanation-and-walkthrough-step-by-step/>

Time analysis: each call to subproblem takes constant time. We call it once from the main routine, and at most twice every time we fill in an entry of array L. There are (m+1)(n+1) entries, so the total number of calls is at most 2(m+1)(n+1)+1 and the time is O(mn).

As usual, this is a worst case analysis. The time might sometimes better, if not all array entries get filled out. For instance if the two strings match exactly, we'll only fill in diagonal entries and the algorithm will be fast.

**Solution 3: DP (10 min)**

**Style 1: Directly convert from Native DFS to 2D DP**

class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        int m = text1.length();

        int n = text2.length();

        // dp[i][j] means the longest common subsequence for position from

        // i till (m - 1) in text1 and poistion from j till (n - 1) in text2

        int[][] dp = new int[m + 1][n + 1];

        // dp[m][0] and dp[0][n] no need initialize since default as 0

        for(int i = m - 1; i >= 0; i--) {

            for(int j = n - 1; j >= 0; j--) {

                if(text1.charAt(i) == text2.charAt(j)) {

                    dp[i][j] = 1 + dp[i + 1][j + 1];

                } else {

                    dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);

                }

            }

        }

        return dp[0][0];

    }

}

Time Complexity : O(NM)

Space Complexity : O(NM)

**Refer to chatGPT**

public class Solution {

public int longestCommonSubsequence(String text1, String text2) {

int m = text1.length();

int n = text2.length();

// Create a DP table with size (m+1) x (n+1) to store the results.

int[][] dp = new int[m + 1][n + 1];

// Fill the DP table.

for (int i = m - 1; i >= 0; i--) {

for (int j = n - 1; j >= 0; j--) {

if (text1.charAt(i) == text2.charAt(j)) {

// If characters match, take the diagonal value and add 1.

dp[i][j] = 1 + dp[i + 1][j + 1];

} else {

// If not, take the maximum of either skipping a character from text1 or text2.

dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);

}

}

}

// The top-left corner of the table will contain the answer for the entire strings.

return dp[0][0];

}

}

**Explanation:**

Dynamic Programming Table (**dp)**:

We create a dp table where dp[i][j] represents the length of the longest common subsequence between text1[i:] (i.e., from i to the end of text1) and text2[j:] (i.e., from j to the end of text2).

Filling the DP Table:

We loop from the last character of text1 and text2 backwards towards the first.

If the characters match (text1[i] == text2[j]), the result is 1 + dp[i+1][j+1], meaning we extend the length of the common subsequence by 1 and move diagonally.

If the characters do not match, the value is the maximum of skipping one character from either text1 or text2, so dp[i][j] = Math.max(dp[i+1][j], dp[i][j+1]).

Base Case:

The extra row and column filled with zeros (i.e., dp[m][n]) represent the base case where one of the strings is empty, so the longest common subsequence is 0.

Time and Space Complexity:

Time complexity: O(m \* n) where m and n are the lengths of the two strings.

Space complexity: O(m \* n) for the DP table.

This 2D DP solution efficiently computes the longest common subsequence length in a bottom-up manner.

**Style 2:**

class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        int m = text1.length();

        int n = text2.length();

        // dp[i][j] means the longest common subsequence for position

        // till (i - 1) in text1 and poistion till (j - 1) in text2

        int[][] dp = new int[m + 1][n + 1];

        for(int i = 1; i <= m; i++) {

            for(int j = 1; j <= n; j++) {

                if(text1.charAt(i - 1) == text2.charAt(j - 1)) {

                    dp[i][j] = 1 + dp[i - 1][j - 1];

                } else {

                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);

                }

            }

        }

        return dp[m][n];

    }

}

Time Complexity : O(NM)

Space Complexity : O(NM)

**Refer to**

<https://leetcode.com/problems/longest-common-subsequence/solutions/598508/python-dp-solution-with-explanation-thinking-process-diagram/>

This is a dynamic programming problem.

First string: str1, Second string: str2.

Assume you are at position ptr1 for str1 and position ptr2 for str2.

Step 1: Compare str1[ptr1] and str[ptr2]. There can be two cases arising from this comparison:

Case 1: They are the same

If they are the same, it is simple, you know that there is 1 good subsequence that you can add up to your answer. Then you increment ptr1 and ptr2. Go back to Step 1.

Case 2: They are not the same.

If that's the case, now you have to explore two options:

Op 1: You increment ptr1, and go back to Step 1.

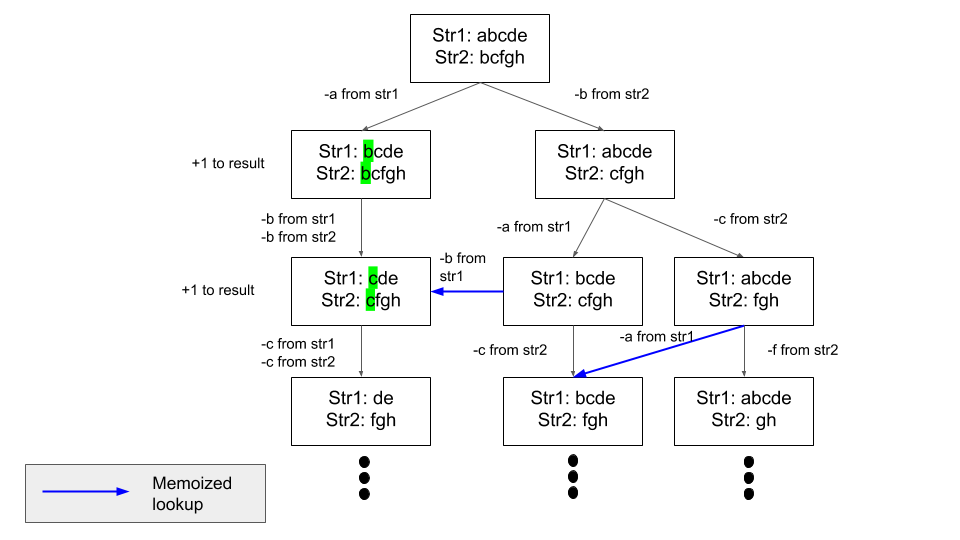
Op 2: You increment ptr2, and go back to Step 1.

You do this until either of your pointers reach to the end of either of the strings.

**Where does Dynamic Programming occur?**

For either of the options in Case 2, there will be a lot overlapping cases. Blue lines in the image below means accessing the memoized results, this is the result of using Dynamic Programming.

In the diagram, I say stuff like '-a from str1' and '-b from str 2'. It is just saying: "remove a from str1 and remove b from str2", respectively. It's similar to incrementing the pointers of respective strings.



In Python, there is an easy way to memoize a function call. For example, if you call a function func(a,b) with same a and b over and over again, the decorator @lru\_cache from functools can memoize this function call with corresponding result so that you don't need to compute over and over again. Internally, it is just creating a hashmap with function parameters as key and returned value as value. LRU Cache Documetation.

**Python Implementation**

from functools import lru\_cache

class Solution:

def longestCommonSubsequence(self, text1: str, text2: str) -> int:

@lru\_cache(maxsize=None)

def memo\_solve(ptr1, ptr2):

if ptr1 == len(text1) or ptr2 == len(text2):

return 0

# Case 1

if text1[ptr1] == text2[ptr2]:

return 1 + memo\_solve(ptr1+1, ptr2+1)

# Case 2

else:

return max(memo\_solve(ptr1+1, ptr2), memo\_solve(ptr1,ptr2+1))

# ^ # ^ Case 2 - Option 1 ^ Case 2 - Option 2

# | \_\_You want the max() result from resulting branches in the tree

return memo\_solve(0,0) # Start the recursion stack from str1[0] and str2[0]

**Refer to**

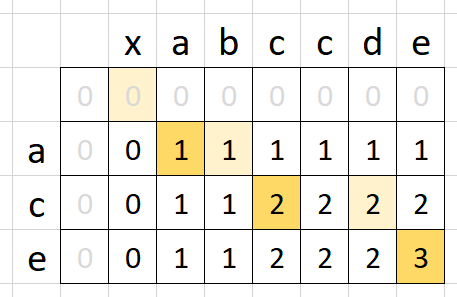
<https://leetcode.com/problems/longest-common-subsequence/solutions/590781/from-brute-force-to-dp/>

<https://leetcode.com/problems/longest-common-subsequence/solutions/348884/c-with-picture-o-nm/>

Bottom-up DP utilizes a matrix m where we track LCS sizes for each combination of i and j.

If a[i] == b[j], LCS for i and j would be 1 plus LCS till the i-1 and j-1 indexes.

Otherwise, we will take the largest LCS if we skip a character from one of the string (max(m[i - 1][j], m[i][j - 1]).This picture shows the populated matrix for "xabccde", "ace" test case.



class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        int[][] dp = new int[text1.length() + 1][text2.length() + 1];

        for (int i = 1; i <= text1.length(); i++) {

            for (int j = 1; j <= text2.length(); j++) {

                if (text1.charAt(i - 1) == text2.charAt(j - 1))

                    dp[i][j] = 1 + dp[i - 1][j - 1];

                else

                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);

            }

        }

        return dp[text1.length()][text2.length()];

    }

}

**Solution 4: DP with Space Optimization (60 min)**

class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        int m = text1.length();

        int n = text2.length();

        // dp[j] stores the length of the longest common subsequence for

        // position till i in text1 and position till (j - 1) in text2.

        // It is similar to what dp[i][j] stored in previous approach.

        int[] dp = new int[n + 1];

        // dpPrev[j] stores the length of  the longest common subsequence

        // for position till (i - 1) in text1 and position till (j - 1) in

        // text2. It is analogous to dp[i - 1][j] in the previous approach.

        int[] dpPrev = new int[n + 1];

        for(int i = 1; i <= m; i++) {

            for(int j = 1; j <= n; j++) {

                if(text1.charAt(i - 1) == text2.charAt(j - 1)) {

                    dp[j] = dpPrev[j - 1] + 1;

                } else {

                    dp[j] = Math.max(dpPrev[j], dp[j - 1]);

                }

            }

            dpPrev = dp.clone();

        }

        return dpPrev[n];

    }

}

Time Complexity : O(NM)

Space Complexity : O(min(N,M))

**Refer to L516. Longest Palindromic Subsequence**

<https://leetcode.com/problems/longest-palindromic-subsequence/editorial/>

**Dynamic Programming with Space Optimization for L1143. Longest Common Subsequence**

**Intuition**

The state transition, as we discussed in previous approaches, is:

dp[i][j] means the longest common subsequence for position till (i - 1) in text1 and position till (j - 1) in text2

class Solution {

    public int longestCommonSubsequence(String text1, String text2) {

        int m = text1.length();

        int n = text2.length();

        // dp[i][j] means the longest common subsequence for position

        // till (i - 1) in text1 and poistion till (j - 1) in text2

        int[][] dp = new int[m + 1][n + 1];

        for(int i = 1; i <= m; i++) {

            for(int j = 1; j <= n; j++) {

                if(text1.charAt(i - 1) == text2.charAt(j - 1)) {

                    dp[i][j] = 1 + dp[i - 1][j - 1];

                } else {

                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);

                }

            }

        }

        return dp[m][n];

    }

}

1. If text1[i - 1] == text2[j - 1], perform dp[i][j] = 1 + dp[i - 1][j - 1];

2. Otherwise, perform dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]).

If we look closely at this transition, to fill dp[i][j] for a particular i and all possible values of j, we only need the values from the current and previous rows. To fill row i in the dp grid, we need the values from row i - 1 (dp[i - 1][j - 1]) and previously computed value in the ith row itself (dp[i][j - 1]). Values in rows i + 1, i + 2, and so on are no longer needed.

Our task is complete if we can store the values of the previous iteration, i.e., for row i + 1 after each iteration of the outer loop.

We can solve this by using two 1D arrays of size n, dp and dpPrev, where n is the length of text2, we don't define array size based off length of text1 (m) is because during downgrading 2D DP array into 1D DP array, the elimination part is exactly the first dimension of 2D DP array, which is the length of text1 (m). We repeat the previous approach by running two loops. The outer loop runs from i = 1 to i = m (text1 length), and the inner loop runs from j = 1 to j = n (text2 length).

dp[i][j] => dp[j]

dp[i - 1][j] => dpPrev[j]

Now, dp[j] stores the length of the longest common subsequence for position till i in text1 and position till (j - 1) in text2. It is similar to what dp[i][j] stored in previous approach.

The other array dpPrev is important to understand. It helps us by remembering the previous state that we completed previously. dpPrev[j] stores the length of  the longest common subsequence for position till (i - 1) in text1 and position till (j - 1) in text2. It is analogous to dp[i - 1][j] in the previous approach.

Because dpPrev stores the answers of substrings length till index (i - 1) and dp stores the answers of substrings length till index i, we must copy the elements of dp to dpPrev after iterating over all the substrings length till index i to prepare for the next iteration. After we copy dp to dpPrev, for the next iteration which considers substrings length till (i + 1), dpPrev will hold values of substrings length till index i which is exactly what we want.

**Refer to**

<https://leetcode.com/problems/longest-common-subsequence/solutions/3498142/best-c-solution-with-5-approaches-with-and-without-dp/>

class Solution {

public:

    int longestCommonSubsequence(string text1, string text2) {

        int m= text1.size();

        int n=text2.size();

        vector<int>prev(n+1, 0);

        vector<int>curr(n+1, 0);

        for(int i=1;i<=m;i++){

            for(int j=1;j<=n;j++){

                if(text1[i-1]==text2[j-1]){

                    curr[j]=1+prev[j-1];

                }

                else{

                    curr[j]=max(curr[j-1], prev[j]);

                }

            }

            prev=curr;

        }

        return prev[n];

    }

};

**Another style**

<https://leetcode.com/problems/longest-common-subsequence/solutions/436719/python-very-detailed-solution-with-explanation-and-walkthrough-step-by-step/>

class Solution:

    def longestCommonSubsequence(self, s1: str, s2: str) -> int:

        m = len(s1)

        n = len(s2)

        if m < n:

            return self.longestCommonSubsequence(s2, s1)

        memo = [[0 for \_ in range(n + 1)] for \_ in range(2)]

        for i in range(m):

            for j in range(n):

                if s1[i] == s2[j]:

                    memo[1 - i % 2][j + 1] = 1 + memo[i % 2][j]

                else:

                    memo[1 - i % 2][j + 1] = max(memo[1 - i % 2][j], memo[i % 2][j + 1])

        return memo[m % 2][n]

<https://leetcode.com/problems/longest-common-subsequence/solutions/348884/c-with-picture-o-nm/>

You may notice that we are only looking one row up in the solution above. So, we just need to store two rows.

int longestCommonSubsequence(string &a, string &b) {

    short m[2][1000] = {};

    for (int i = 0; i < a.size(); ++i)

        for (int j = 0; j < b.size(); ++j)

            m[!(i % 2)][j + 1] = a[i] == b[j] ? m[i % 2][j] + 1 : max(m[i % 2][j + 1], m[!(i % 2)][j]);

    return m[a.size() % 2][b.size()];

}

**Complexity Analysis**

Time: O(nm), where n and m are the string sizes.

Memory: O(min(n,m)), assuming that we will use a smaller string for the column dimension.

**Refer to**

[L516.Longest Palindromic Subsequence](note://9BDFB43866AE4C9D8D4278698105F864)

[L583.Delete Operation for Two Strings (Ref.L712,L72,L1143)](note://WEB12ebad84cb577589b9576d7f802df5ee)

[L712.Minimum ASCII Delete Sum for Two Strings (Ref.L72,L583,L1143)](note://WEBa910883e40bb3571527f8753e2b859be)